CS 595 - Hot topics in database systems: **Data Provenance**

I. Database Provenance

I.1 Provenance Models and Systems

Boris Glavic

November 14, 2012

Outline

Why Provenance Querying?

- Size of provenance information
- Extract parts of provenance that are interesting/important
  - Access to both data and provenance
  - ⇒Simple form is forward/backward queries
  - Find patterns or paths
  - Need be able to aggregate, correlate, filter

ILLINOIS INSTITUTE OF TECHNOLOGY

Challenges for Querying Provenance

- Different model for data and provenance
  - Similar problems as for provenance compression and indexing
  - Existing query language may not be used
- Queries that span regular data and provenance
  - Need features of data query language
  - + Provenance query features such as backward and forward queries

ILLINOIS INSTITUTE OF TECHNOLOGY

Query Languages with Provenance Support

- Language features for retrieving provenance
- ≠ Language features for controlling provenance generation
- ⇒**Retrieval** or **Generation** language with respect to provenance

Example (DBNotes' Query Language *pSQL*)

- *pSQL* has language features to control how annotations are propagated
  - ⇒**Generation** language
- Accessing annotations for, .e.g., filtering
  - ⇒Also a **Retrieval** language

Approaches for Provenance Query Languages

Relation to Data Query Language

- Extending the Data Query Language $\mathcal{Q}$
  - Add new provenance-specific features to $\mathcal{Q}$
  - E.g.: Perm, DBNotes, Orchestra, WHIPS
- Create a new language from scratch
  - Adapt to what is needed for provenance querying
  - E.g.: ProQL, Some Workflow approaches

Implementation

- Compile into standard queries of $\mathcal{Q}$ (Rewrite)
  - E.g.: Perm, DBNotes, Orchestra, WHIPS
- Create new or modified execution engine for these queries
  - E.g.: Ariadne, Workflow-Approaches

Extending the Data Query Language

Discussion

- Reuse of common functionality
  - Important for powerful query languages like SQL or XQuery
- Query paradigm may not be good fit for provenance
  - E.g., as we have seen for interval encoding
- Integration of querying data with querying or generating provenance
  - Retrieval: Easier to express queries that access both provenance and data properties
  - Generation: Use the query language to select for which data we want provenance

Developing a New Query Language

Discussion

- Fine-tuned for provenance queries
  - Limit to what is strictly necessary for provenance querying
- Functionality for querying data has to be rebuild from scratch
  - ⇒likely less powerful and stable as language such as SQL

ILLINOIS INSTITUTE OF TECHNOLOGY

Implementation: Rewrite

Discussion

- Reuse Existing Execution Engines
  - Can be critical for systems like databases
- Performance may be impacted negatively
  - Operations need to express provenance queries not supported efficiently
- Full integration with data query language

ILLINOIS INSTITUTE OF TECHNOLOGY

Implementation: Modify Execution Engine

Discussion

- Can add new operations that speed up provenance access patterns
  - E.g., path-queries for relational data or computing the provenance of an aggregation
- Integration with data query language
  - Execution engine already supports data query language
- May require significant changes to the engine
  - Because of different provenance data model
  - Optimization may be different for provenance operations

ILLINOIS INSTITUTE OF TECHNOLOGY

Implementation: Build Execution Engine

Discussion

- Optimized for provenance query language
- No support for data query language
  - Either no integration
  - or build execution engine from scratch
- Naturally limited scope
  - To time intensive to rebuild DBMS from scratch

ILLINOIS INSTITUTE OF TECHNOLOGY

Query Languages Overview

We will cover

- pSQL (DBNotes)
- SQL-PLE (Perm)
- ProQL (Orchestra)
- Color algebra (Mondrian)

ILLINOIS INSTITUTE OF TECHNOLOGY

Outline

pSQL

DBNotes Review

- DBMS with support for annotations
- Implemented as middle-ware over standard DBMS
- Each attribute value in DB carries set of (textual) annotations
- Annotations are propagated during query execution
    - Where-provenance, if every attribute value is annotated with an identifier R, t, a (relation R, tuple t, attribute a)
- Query language is **pSQL**

pSQL Syntax

pSQL query block

```
SELECT DISTINCT  selectlist
FROM             fromlist
WHERE            wherelist
PROPAGATE        (DEFAULT
                  | DEFAULT-ALL
                  | R1.a1 TO b1, ..., Rn.an TO bn)
```

- pSQL query is union of query blocks
- *selectlist*: Select attributes and rename ($R_1.a_1 \rightarrow b_1$)
- *fromlist*: List of relations with potential aliases and ANNOT expressions
- *wherelist*: conjunction of attribute-attribute or attribute-constant equalities

pSQL Language Overview

Language Features

- pSQL is mostly a provenance **generation** language
  - Generation (annotation propagation) is always active
    - Annotations are propagated for every query
  - `PROPAGATE` clause determines how annotations are propagated
- pSQL has provenance **retrieval** features
  - Access annotations as data using `ANNOT` in from clause

Implementation

- **Rewrite** into SQL

Language Type

- **Extension of data query language**: Subset of SQL

pSQL - the PROPAGATE clause

- Defines how annotations are copied from input to output
- Three different options:
  - Custom
  - `DEFAULT`
  - `DEFAULT-ALL`

pSQL - the PROPAGATE clause

- Defines how annotations are copied from input to output
- Three different options:
  - Custom
  - DEFAULT
  - DEFAULT-ALL

Custom

- User defines annotation propagation at the query level
- $R_1.a_1$ TO b_1 = add annotations from input attribute $R_1.a_1$ to result attribute b_1.

ILLINOIS INSTITUTE OF TECHNOLOGY

pSQL - the PROPAGATE clause

- Defines how annotations are copied from input to output
- Three different options:
  - Custom
  - `DEFAULT`
  - `DEFAULT-ALL`

DEFAULT

- Propagate annotations from all cells in the Where-provenance

ILLINOIS INSTITUTE OF TECHNOLOGY

pSQL - the PROPAGATE clause

- Defines how annotations are copied from input to output
- Three different options:
  - Custom
  - `DEFAULT`
  - `DEFAULT-ALL`

DEFAULT-ALL

- Propagate annotations from all cells in the insensitive Where-provenance

pSQL example query

Example

```
SELECT P.name, A.street, A.city
FROM person P, address A
WHERE P.addr = A.street
PROPAGATE P.name TO name,
          P.addr TO street,
          P.addr TO city
```

person

	name	addr
p_1	Bob	10 W 31st$^{\{a_1\}}$
p_2	Alice$^{\{a_2,a_3\}}$	75 Cermak
p_2	Gertrud	15 Ellis

address

	street	city
a_1	10 W 31st	Chicago$^{\{a_4\}}$
a_2	75 Cermak$^{\{a_5\}}$	Chicago
a_3	15 Ellis	Berlin

pSQL example query

Example

result

	name	street	city
t_1	Bob	10 W 31st$^{\{a_1\}}$	Chicago$^{\{a_1\}}$
t_2	Alice$^{\{a_2,a_3\}}$	75 Cermak	Chicago
t_3	Gertrud	15 Ellis	Berlin

person

	name	addr
p_1	Bob	10 W 31st$^{\{a_1\}}$
p_2	Alice$^{\{a_2,a_3\}}$	75 Cermak
p_2	Gertrud	15 Ellis

address

	street	city
a_1	10 W 31st	Chicago$^{\{a_4\}}$
a_2	75 Cermak$^{\{a_5\}}$	Chicago
a_3	15 Ellis	Berlin

OF TECHNOLOGY

Accessing annotations

ANNOT expression

- `ANNOT(R.a) alias`
  - relation R and attribute a
  - `alias` assigns a name to the annotation expression
- Use in `FROM` clause
- `ANNOT`: access set of annotations for attribute $R.a$.

Example

person

	name	addr
p_1	Bob	10 W 31st$^{\{a_1\}}$
p_2	Alice$^{\{a_2,a_3\}}$	75 Cermak
p_2	Gertrud	15 Ellis

```
\lstset{morekeywords=ANNOT}
SELECT P.name, P.addr
FROM person P,
     ANNOT(P.name) a
WHERE a LIKE '%dumb%'
PROPAGATE a TO name
          AND P.addr TO addr
```

ANNOT semantics

Sketch Query Evaluation

- Build cross-product of relations and annotation sets in `FROM`
- Apply `WHERE` condition to result
- Evaluate `PROPAGATE` clause for remaining annotations

Semantics

- If `ANNOT(R.a) A` is only used in `WHERE`
  - ⇒Works as existential check for annotation that fulfill `WHERE`
- If `ANNOT(R.a) A` is also used in `PROPAGATE`
  - ⇒only propagate annotations that fulfill `WHERE`

OF TECHNOLOGY

ANNOT Examples

Example (Only WHERE)

person

	name	addr
p_1	Bob	10 W 31st$^{\{a_1\}}$
p_2	Alice$^{\{a_2,a_3\}}$	75 Cermak
p_2	Gertrud	15 Ellis

```
SELECT P.name, P.addr
FROM person P,
     ANNOT(P.name) a
WHERE a LIKE '%dumb%'
PROPAGATE P.name TO name
          AND P.addr TO addr
```

ILLINOIS INSTITUTE OF TECHNOLOGY

ANNOT Examples

Example (WHERE and PROPAGATE)

person

	name	addr
p_1	Bob	10 W 31st$^{\{a_1\}}$
p_2	Alice$^{\{a_2,a_3\}}$	75 Cermak
p_2	Gertrud	15 Ellis

```
\lstset{morekeywords=ANNOT}
SELECT P.name, P.addr
FROM person P,
     ANNOT(P.name) a
WHERE a LIKE '%dumb%'
PROPAGATE a TO name
          AND P.addr TO addr
```

ILLINOIS INSTITUTE OF TECHNOLOGY

pSQL Implementation

- Use relational storage schema for annotated relations
- **Rewrite** pSQL into SQL queries over that schema

Storage Scheme

- Schema
  - For each attribute A in relation R
  - Add attribute A_a that stores annotations on A
- Instance
  - Each tuple can store one annotation on each attribute
  - Duplicate tuples to fit in more annotations

Example

- $R(a, b)$ will be $R(a, a_a, b, b_a)$

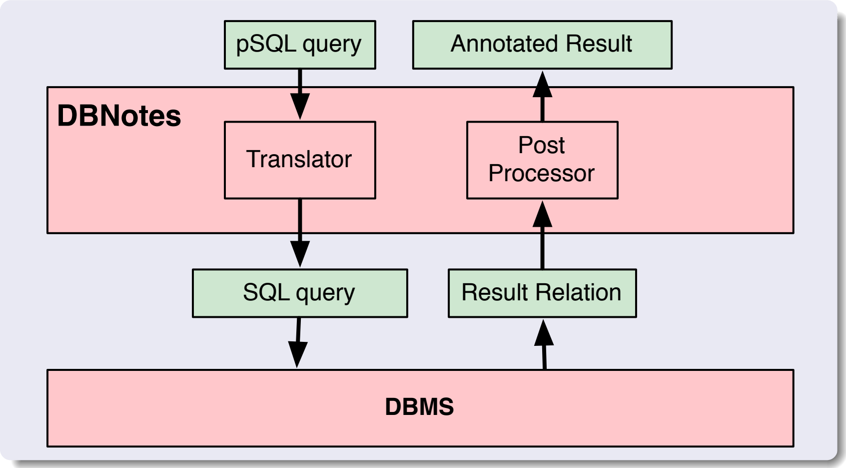
R

	a	b
r_1	$1^{a_1,a_2}$	2^{a_3}
r_2	2^{a_4}	3

R'

	a	a_a	b	b_a
r_1	1	a_1	2	a_3
r_1	1	a_2	2	-
r_1	2	a_4	3	-

Middleware Implementation Architecture (DBNotes)



Translator

1. Transform propagate clauses into *Custom* form
2. Build bins for each output attribute b
   - Add $R.a$ to bin if $R.a$ `TO` b in propagate clause

Translator

1. Transform propagate clauses into *Custom* form
2. Build bins for each output attribute b
   - Add $R.a$ to bin if $R.a$ `TO` b in propagate clause
3. Generate intermediate queries Q_1 to Q_n
   - While at least on bin not empty
   - Take one attribute a from each bin
   - Generate query that projects each a_a to b_a
   - Use `NULL TO` b_a if bin is empty

ILLINOIS INSTITUTE OF TECHNOLOGY

Translator

1. Transform propagate clauses into *Custom* form
2. Build bins for each output attribute b
   - Add $R.a$ to bin if $R.a$ `TO` b in propagate clause
3. Generate intermediate queries Q_1 to Q_n
   - While at least on bin not empty
   - Take one attribute a from each bin
   - Generate query that projects each a_a to b_a
   - Use `NULL TO` b_a if bin is empty
4. Generate wrapper query
   - *orderlist* all attribute of pSQL query

```
SELECT DISTINCT *
FROM (Q1 UNION ... UNION Qn)
ORDER BY orderlist
```

Post-Processor

- Gather all annotations for each result tuple
- Works like aggregation
  1. Initialize each attribute annotation sets to empty set
  2. For each tuple add annotations to sets
  3. If tuple has different attribute value
     - Output annotated result tuple
     - Start over at (1)

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Processing

```
SELECT P.name, A.street, A.city
FROM person P, address A
WHERE P.addr = A.street
PROPAGATE P.name TO name,
          P.addr TO street,
          P.addr TO city
```

person

	name	addr
p_1	Bob	10 W 31st$^{\{a_1\}}$
p_2	Alice$^{\{a_2,a_3\}}$	75 Cermak
p_2	Gertrud	15 Ellis

address

	street	city
a_1	10 W 31st	Chicago$^{\{a_4\}}$
a_2	75 Cermak$^{\{a_5\}}$	Chicago
a_3	15 Ellis	Berlin

OF TECHNOLOGY

Example Processing - Storage

person′

	name	$name_a$	addr	$addr_a$
p_1	Bob		10 W 31st	a_1
p_2	Alice	a_2	75 Cermak	
p_2	Alice	a_3	75 Cermak	
p_2	Gertrud		15 Ellis	

address′

	street	$street_a$	city	$city_a$
a_1	10 W 31st		Chicago	a_4
a_2	75 Cermak	a_5	Chicago	
a_3	15 Ellis		Berlin	

Example Processing - Intermediate Queries

- **Bin** name: {P.name}
- **Bin** street: {P.addr}
- **Bin** city: {P.addr}

```
SELECT P.name, P.name_a,
       A.street, P.addr_a AS street_a,
       A.city, P.addr_a AS city_a
FROM person' P, address' A
WHERE P.addr = A.street
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Processing - Wrapper Query

```
SELECT DISTINCT *
FROM (SELECT P.name, P.name_a,
        A.street, P.addr_a AS street_a,
        A.city, P.addr_a AS city_a
      FROM person' P, address' A
      WHERE P.addr = A.street) AS i
ORDER BY name, street, city
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Processing - Post-processing

result

	name	$name_a$	street	$street_a$	city	$city_a$
t_1	Bob		10 W 31st	a_1	Chicago	a_1
t_2	Alice	a_2	75 Cermak		Chicago	
t_2	Alice	a_3	75 Cermak		Chicago	
t_3	Gertrud		15 Ellis		Berlin	

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Processing - Post-processing

result

	name	street	city
t_1	Bob	10 W 31st$^{\{a_1\}}$	Chicago$^{\{a_1\}}$
t_2	Alice$^{\{a_2,a_3\}}$	75 Cermak	Chicago
t_3	Gertrud	15 Ellis	Berlin

result

	name	name_a	street	street_a	city	city_a
t_1	Bob		10 W 31st	a_1	Chicago	a_1
t_2	Alice	a_2	75 Cermak		Chicago	
t_2	Alice	a_3	75 Cermak		Chicago	
t_3	Gertrud		15 Ellis		Berlin	

Translation for ANNOT Expressions

- For each `ANNOT(R.a) A` where R has attributes $a_1, \ldots a_n$

1. Add query to `FROM` clause
   - `(SELECT R.`a_a`, R.* FROM` R' `) AS A`
2. Transform references to A in `WHERE` into $A.a_a$
3. Add join condition $R.a_1 = A.a_1 \ldots R.a_n = A.a_n$ to `WHERE`
4. Apply transformation for `PROPAGATE`

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Translation

Example

```
SELECT P.name, P.addr
FROM person P,
     ANNOT(P.name) a
WHERE a LIKE '%dumb%'
PROPAGATE P.name TO name
          AND P.addr TO addr
```

⇒

```
SELECT DISTINCT *
FROM (SELECT P.name, A.name_a, P.addr, P.addr_a
     FROM person' P,
          (SELECT name_a, name, addr FROM person') AS A
     WHERE P.name = A.name AND P.addr = A.addr
           AND A.name_a LIKE '%dumb%') AS i
ORDER BY name, addr
```

Discussion pSQL

- **Extension of existing query language** (SQL)
- For provenance **Generation** and **Retrieval**
- Implemented as **Rewrite**
- Provenance generation always active

Outline

SQL-PLE

Perm Recap

- DBMS with support for provenance generation
- Provenance is generated on-demand
- Implemented as extension of PostgreSQL
- Provenance represented as regular relational data
- Provenance features through **SQL-PLE**

ILLINOIS INSTITUTE OF TECHNOLOGY

SQL-PLE Language Overview

Language Features

- SQL-PLE = provenance language extension
- **Generation**: Generate provenance for a query
  - ...according to input relations
  - ...intermediate results
  - Support for external provenance
- **Retrieval**
  - SQL as query language for data and provenance
  - works because of relational representation of provenance
- Full SQL integration
  - Provenance generation and retrieval generate standard relations
  - ⇒Used like any other SQL query construct

SQL-PLE Language Overview

Implementation

- **Rewrite** into SQL

Language Type

- **Extension of data query language**: super-set of SQL

ILLINOIS INSTITUTE OF TECHNOLOGY

Language Features - Provenance Generation

Request Provenance

```
SELECT prov_expr select_clause
FROM from_list
WHERE where_expr
GROUP BY group_by
HAVING having_expr
ORDER BY order_expr
```

- prov_expr = [PROVENANCE [ON CONTRIBUTION (@cs_type@)]]
  - If present, compute provenance in addition to query results
- cs_type = Which provenance model should be used
  - PI-CS (Perm Influence)
  - C-CS (Perm Copy similar to Where for tuples)
  - Where
  - How (Provenance Polynomials)

Language Features - Provenance Generation

Trace Back to Intermediate Results

- Default: compute witness-lists for leafs of query tree
- Append BASERELATION to a FROM clause entry
- Use result of the subtree under this item instead of leafs of the subtree

Example

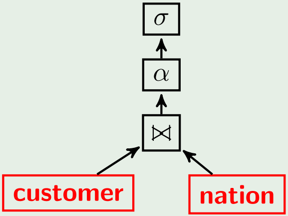
```
SELECT PROVENANCE *
FROM (SELECT count(*) AS numCust, country
      FROM customer c,
           nation n
      WHERE c.nId = n.Id
      GROUP BY country) BASERELATION AS cCount
WHERE country = 'USA'
```

Language Features - Provenance Generation

Trace Back to Intermediate Results

- Default: compute witness-lists for leafs of query tree
- Append `BASERELATION` to a `FROM` clause entry
- Use result of the subtree under this item instead of leafs of the subtree

Example

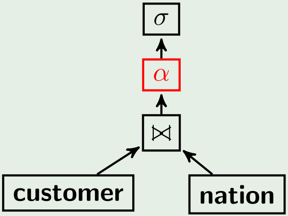


Language Features - Provenance Generation

Trace Back to Intermediate Results

- Default: compute witness-lists for leafs of query tree
- Append `BASERELATION` to a `FROM` clause entry
- Use result of the subtree under this item instead of leafs of the subtree

Example



Language Features - Provenance Generation

External and User-Defined Provenance

- Default: All attributes of relation are used as provenance
- Append `PROVENANCE (A)` to `FROM` clause item
- Attributes in list A are used as provenance
- Use cases
  - External provenance: manually created or from other provenance-aware system
  - Concise representation: Use tuple identifiers instead of complete tuples

ILLINOIS INSTITUTE OF TECHNOLOGY

Language Features - Provenance Generation

Example

```
SELECT PROVENANCE *
FROM (SELECT count(*) AS numCust, country
      FROM customer PROVENANCE (cId) AS c,
           nation PROVENANCE (nId) AS n
      WHERE c.nId = n.nId
      GROUP BY country) AS cCount
WHERE country = 'USA'
```

Language Features - Provenance Generation

Example

```
SELECT PROVENANCE *
FROM (SELECT count(*) AS numCust, country
      FROM customer c,
           (SELECT n.*, h.description
           FROM nation n, nHistory h)
           PROVENANCE (description) AS n
      WHERE c.nId = n.Id
      GROUP BY country) AS cCount
WHERE country = 'USA'
```

ILLINOIS INSTITUTE OF TECHNOLOGY

SQL-PLE Implementation Overview

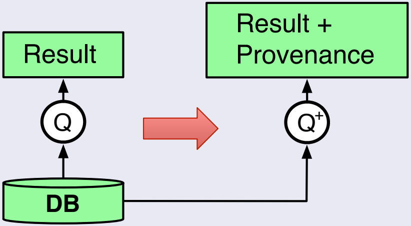
- **Rewrite**
- Rewrite new language features and provenance generation into plain SQL
- Uses algebraic rewrite rules for provenance generation
- Was discussed in the provenance models class on Perm

ILLINOIS INSTITUTE OF TECHNOLOGY

Query Rewrite

Approach

- Rewrite query $Q \rightarrow$ query Q^+
  - Q^+ computes provenance $+$ original results of Q
  - by adding provenance to the inputs (duplicate attributes)
  - propagates provenance through the operations of the query



Query Rewrite

Approach

- Rewrite query $Q \rightarrow$ query Q^{+}
  - Q^{+} computes provenance + original results of Q
  - by adding provenance to the inputs (duplicate attributes)
  - propagates provenance through the operations of the query
- Rewrite defined as rules
  - One rule for each "type of operation"
  - Query rewritten by recursive application to each operation

ILLINOIS INSTITUTE OF TECHNOLOGY

Query Rewrite Cont.

Advantages

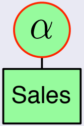
- Q^+ is single standard SQL
  - ⇒Reuse DB technology
- Q^+ accesses the same data as Q
  - ⇒No need to store extra information
- Provenance generation as orthogonal SQL extension
  - ⇒Queries over provenance benefit from optimizer
- Correctness formally proven

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Query Rewrite

Example Query

```
SELECT sum(revenue) AS total, shop
FROM sales
GROUP BY shop;
```



Example (DB + Query Result)

Sales

shop	month	revenue
Migros	Jan	100
Migros	Feb	10
Migros	Mar	10
Coop	Jan	25
Coop	Feb	25

Result

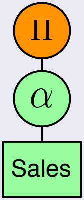
total	shop
120	Migros
50	Coop

Example Query Rewrite

Rewrite: 1

Example Query

```
SELECT total, shop, P(Q+)
FROM
    (SELECT sum(revenue) AS total, shop
    FROM sales GROUP BY shop) AS orig
```



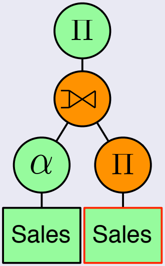
ILLINOIS INSTITUTE OF TECHNOLOGY

Example Query Rewrite

Rewrite: 2

Example Query

```
SELECT total, shop, P(Q+)
FROM
    (SELECT sum(revenue) AS total, shop
    FROM sales GROUP BY shop) AS orig
    LEFT OUTER JOIN
    (SELECT shop AS shop', P(orig+)
    FROM sales+
    ) AS orig+
    ON (shop = shop');
```



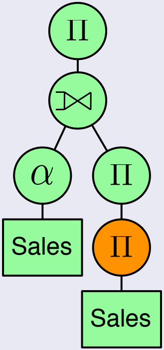
ILLINOIS INSTITUTE OF TECHNOLOGY

Example Query Rewrite

Rewrite: 3 Provenance Attrs: 1

Example Query

```
SELECT total, shop, P(Q+)
FROM
    (SELECT sum(revenue) AS total, shop
    FROM sales GROUP BY shop) AS orig
    LEFT OUTER JOIN
    (SELECT shop AS shop′, P(orig+)
    FROM
        (SELECT shop, month, revenue,
                shop AS P(shop),
                month  AS  P(month),
                revenue AS  P(revenue)
        FROM sales) AS sales+
    ) AS orig+
    ON (shop = shop′);
```



Example Query Rewrite

Rewrite: 3 Provenance Attrs: 2

Example Query

```
SELECT total, shop, P(Q+)
FROM
    (SELECT sum(revenue) AS total, shop
    FROM sales GROUP BY shop) AS orig
    LEFT OUTER JOIN
    (SELECT shop AS shop′, P(shop), P(month), P(revenue)
    FROM
        (SELECT shop, month, revenue,
                shop AS P(shop), month AS P(month),
                revenue AS P(revenue)
        FROM sales) AS sales+
    ) AS orig+
    ON (shop = shop′);
```

Example Query Rewrite

Rewrite: 3 Provenance Attrs: 3

Example Query

```
SELECT total, shop, P(shop), P(month), P(revenue)
FROM
    (SELECT sum(revenue) AS total, shop
    FROM sales GROUP BY shop) AS orig
    LEFT OUTER JOIN
    (SELECT shop AS shop′, P(shop), P(month), P(revenue)
    FROM
        (SELECT shop, month, revenue,
                shop AS P(shop), month AS P(month),
                revenue AS P(revenue)
        FROM sales) AS sales+
    ) AS orig+
    ON (shop = shop′);
```

Example Rewritten Query Result

Example (Result + Provenance)

total	shop	P(shop)	P(month)	P(revenue)
120	Migros	Migros	Jan	100
120	Migros	Migros	Feb	10
120	Migros	Migros	Mar	10
50	Coop	Coop	Jan	25
50	Coop	Coop	Feb	25

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Rewritten Query Result

Example (Result + Provenance)

total	shop	P(shop)	P(month)	P(revenue)
120	Migros	Migros	Jan	100
120	Migros	Migros	Feb	10
120	Migros	Migros	Mar	10
50	Coop	Coop	Jan	25
50	Coop	Coop	Feb	25

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Rewritten Query Result

Example (Result + Provenance)

total	shop	P(shop)	P(month)	P(revenue)
120	Migros	Migros	Jan	100
120	Migros	Migros	Feb	10
120	Migros	Migros	Mar	10
50	Coop	Coop	Jan	25
50	Coop	Coop	Feb	25

ILLINOIS INSTITUTE OF TECHNOLOGY

How to Implement Partial Generation

- (q) BASERELATION AS q in FROM clause
- Apply rewrite rule for base relation to q instead of rewriting q

Example

```
SELECT PROVENANCE *
FROM (SELECT count(*) AS numCust, country
      FROM customer c,
           nation n
      WHERE c.nId = n.Id
      GROUP BY country) BASERELATION AS cCount
WHERE country = 'USA'
```

How to Implement Partial Generation

- (q) BASERELATION AS q in FROM clause
- Apply rewrite rule for base relation to q instead of rewriting q

Example

```
SELECT numcust, country,
       numcust AS prov_ccount_numcust,
       country AS prov_ccount_country
  FROM ( SELECT count(*) AS numcust, n.country
          FROM customer c, nation n
         WHERE c.nid = n.id
         GROUP BY n.country) ccount
 WHERE country = 'USA'
```

How to User Defined Provenance

- (q) PROVENANCE (A) AS q in FROM clause
- Apply a modified

Example

```
SELECT PROVENANCE *
FROM (SELECT count(*) AS numCust, country
      FROM customer PROVENANCE (cId) AS c,
           nation PROVENANCE (nId) AS n
      WHERE c.nId = n.nId
      GROUP BY country) AS cCount
WHERE country = 'USA'
```

GY

How to User Defined Provenance

Example

```
SELECT numcust, country, prov_cid, prov_nid
FROM (SELECT orig.numcust, orig.country,
             rewAgg.prov_cid, rewAgg.prov_nid
      FROM (SELECT count(*) AS numcust, n.country
            FROM customer c, nation n
            WHERE c.nOd = n.nId
            GROUP BY n.country) orig
        LEFT JOIN
          (SELECT n.country,
                  c.cId AS prov_cid, n.id AS prov_nid
           FROM customer c, nation n
           WHERE c.nid = n.id) rewAgg
        ON  orig.country = rewAgg.country) ccount
WHERE ccount.country = 'USA'::text
```

Discussion SQL-PLE

- **Extension of existing query language** (SQL)
- For provenance **Generation** and **Retrieval**
- Implemented as **Rewrite**
- Full integration with SQL
- Provenance generation on-demand
- Partial provenance generation and external provenance
- Supports multiple provenance models
- Retrieval using standard SQL

ILLINOIS INSTITUTE OF TECHNOLOGY

Outline

ProQL

Overview

- Query language for graph provenance representation
- Provenance graphs model How-provenance (provenance polynomials) + mappings (views)
- Graphs are stored in relational database
- Language features for evaluating provenance polynomials in a specific semiring
  - Trust semiring
  - Natural numbers semiring (Bag semantics)
- Main motivation - use in Orchestra

OF TECHNOLOGY

ORCHESTRA Recap

Overview

- **C**ollaborative **D**ata **S**haring **S**ystem
- Network of peers
- Each peer has independent schema and instance
- Peers update their instances without restrictions
- Schema mappings define relationships between schemata
  - Can be partial
- Periodically peers trigger exchange of updates based on mappings

Schema mappings

- Schema mapping: Logical constraints that define the relationship between two schemata
- Different schema may store the same information in different structure
- Schema mappings model these structures in the schema relate
- With some extra mechanism can be use to translate data from one schema into the other
- For now consider mappings as views

Example

- **Schema** S_1: `Person(Name, AddrId)`, `Address(Id, City, Street)`
- **Schema** S_2: `LivesAt(Name, City)`

GY

Update Exchange

- Each peer updates its instance as he pleases
- A log of update operations is kept
- Peers can trigger an update exchange

Update Exchange

- Determine updates since last exchange
- Translate updates from peers according to schema mappings
- Eagerly compute provenance during update exchange

ILLINOIS INSTITUTE OF TECHNOLOGY

ProQL Language Overview

Language Features

- ProQL is not a **generation** language
  - No language constructs for generating provenance
- ProQL focuses on **retrieval** features
  - **Graph projection**: select parts from provenance graph
  - **Annotation computation**: evaluate in semi-ring

Implementation

- **Rewrite** into SQL queries over relational graph encoding + metadata

Language Type

- **New language**:
  - **Input**: Provenance graph
  - **Output**: Provenance graph + Var. Bindings + Annotations

Provenance Polynomials Recap

Rationale

- Use semiring annotations to model provenance
- Annotate a query result tuple with the semiring expression that was used to compute it

Provenance Polynomials Semiring

- $(\mathbb{N}[I], +, \times, 0, 1)$
- $\mathbb{N}[I]$ - Polynomials with natural number exponents
  - Variables: One per tuple in I
- Convention: annotate each instance tuple with a variable named after its tuple *id*

GY

The "How" Part

Interpretation of + and ×

- +: Alternative use of tuples
  - Operators: Union, Projection
  - Check set-semantics: only one tuples is need ⇒$\vee$ as + operation
  - Check bag-semantics: multiplicities are additive ⇒natural number addition as +
- ×: Conjunctive use of tuples
  - Operations: Join
  - Check set-semantics: both tuples are needed ⇒$\wedge$ as × operation
  - Check bag-semantics: multiplicities of matching tuples are multiplied ⇒natural number multiplication as ×

Provenance Polynomials Example

Example

$$q = \pi_a(R)$$

R

	a	b
t_1	1	2
t_2	1	3

Q

	a
$t_1 + t_2$	1

ILLINOIS INSTITUTE OF TECHNOLOGY

Provenance Polynomials Example II

Example

$$q = \pi_{Name}(E \bowtie \sigma_{Dep=CS}(P) \bowtie A)$$

Employee

	Id	Name
e_1	1	Peter
e_2	2	Gertrud
e_2	3	Michael

Assigned

	PName	Id
a_1	Server	1
a_2	Server	2
a_3	Webpage	2
a_4	Fire CS	3

Project

	PName	Dep
p_1	Server	CS
p_2	Webpage	CS
p_3	Fire CS	HR

ILLINOIS INSTITUTE OF TECHNOLOGY

Provenance Polynomials Example II

Example

$$q = \pi_{Name}(E \bowtie \sigma_{Dep=CS}(P) \bowtie A)$$

$$(q)(t) = \sum_{u.A=t} E(u.E) \times P(u.P) \times (Dep = CS)(u.P) \times A(u.P)$$

Q

	Name
$e_1 \times a_1 \times p_1$	Peter
$(e_2 \times a_2 \times p_1) + (e_2 \times a_3 \times p_2)$	Gertrud

Employee

	Id	Name
e_1	1	Peter
e_2	2	Gertrud
e_2	3	Michael

Assigned

	PName	Id
a_1	Server	1
a_2	Server	2
a_3	Webpage	2
a_4	Fire CS	3

Project

	PName	Dep
p_1	Server	CS
p_2	Webpage	CS
p_3	Fire CS	HR

Provenance in ORCHESTRA

- Use $\mathbb{N}[I]$ (provenance polynomials)
- Add functions $m_1, \ldots, m_n$ to represent mappings (views)
- E.g., $m_3(m_1(x + yz) + m_2(zu))$ means that tuple in view m_3 was derived from
  - tuple in view m_1 derived from x, y, z
  - tuple in view m_2 derived from z, u

ILLINOIS INSTITUTE OF TECHNOLOGY

Provenance graph

Intuition

- Graph representation of the provenance polynomials
- For sets of tuples and views
- Reuse overlapping parts of provenance

Provenance graph

Structure

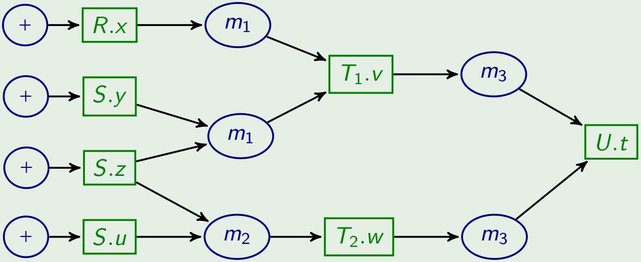
- Directed graph with two types of nodes
  - Nodes representing tuples (rectangles)
  - Nodes representing application of mappings (views)
- View nodes are replicated for each tuple in view result
- An edge from a view node to a tuple node
  - ⇒view generated this tuple
- Incoming edges from tuple node to a view node
  - ⇒view used tuple to generate the output tuple
- + view nodes are connected to "base" tuples

OF TECHNOLOGY

Example Provenance Graph

Example

$$\mathbb{N}[I](t) = m_3(m_1(x + yz) + m_2(zu))$$



Provenance Graph Interpretation

- Alternative derivations (+) as independent view nodes
- Conjunctive derivations ($\times$) = multiple incoming edges
- Cycles are possible (mappings use datalog style recursion)
- Path from tuple t to tuple t'
  - t is in provenance of t'
- No direct edges between tuple nodes and tuple nodes
- No direct edges between view nodes and view nodes

ProQL: Graph Projection

- Determines subgraph of the input provenance graph to return
  - Matching parts of the input graph to **path expressions** of query
- Binds variables to tuple and view nodes
  - Returns tuples of bindings (to nodes) for the variables of the query
  - These nodes are called **distinguished** nodes
- Strict subgraph: no new nodes or paths
- View node returned ⇒also return directly connected tuple nodes

ILLINOIS INSTITUTE OF TECHNOLOGY

Binding Variables

Tuple Expressions

- `TupleExpr: [Relation Variable]`
- both parts are optional
- Convention: Variables start with $ sign
- Variable iterates over all tuples nodes for relation

Example

- [] matches all tuple nodes
- [$x] binds all tuples node to variable $x
- [R] matches all tuples nodes from relation R
- [R$y] binds all tuples nodes from relation R to variable $y

Path Expressions

- Path expressions are of the form:
- `PathExpr: TupleExpr | TuplExpr Op TupleExpr`
- `Op` determines the type of path
- Optionally binds variables to view nodes
- `Op: <- | <-+ | <ViewName | <Var`
- Path expression matches paths between tuples matching the tuple expressions
- Result of path expression is subgraph spanned by the path
- . . . according to `Op`

OF TECHNOLOGY

<- Expressions

- Match paths of length two
- Between two tuple nodes
- Going through any view node

Example

- `[] <- [R]`: Matches an application of any view (view node) that uses a tuple from relation R (incoming edge from tuple node)
- `[R] <- [S]`: Matches view application (view node) that uses a tuple from relation S to derive a tuple from R

<-+ Expression

- Matches paths of arbitrary length
- Between two tuple nodes

Example

- `[R] <-+ [S]`: Matches any path that starts in a tuple node from S and ends in a tuple node from R
- ⇒paths that connect tuples from R with their provenance in S

ILLINOIS INSTITUTE OF TECHNOLOGY

<ViewName Expression

- Matches a path of length two
- Between two tuple nodes
- Going through view node for `ViewName`

Example

- `[] <`m_1 `[]`: Matches any application of view m_1
- ⇒paths that connect tuples derived though view m_1 to their provenance tuples from the input of the view

ILLINOIS INSTITUTE OF TECHNOLOGY

<Var Expression

- Match paths of length two
- Between two tuple nodes
- Going through any view node
- The view node is bound to `Var`

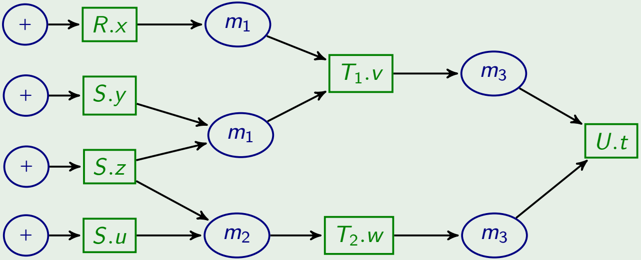
Example

- `[R] <$x []`: Matches an application of any view (view node) that uses a tuple from relation *R* (incoming edge from tuple node)
- ⇒the view node is bound to $x

Example View Expressions

Example

$$\mathbb{N}[I](t) = m_3(m_1(x + yz) + m_2(zu))$$

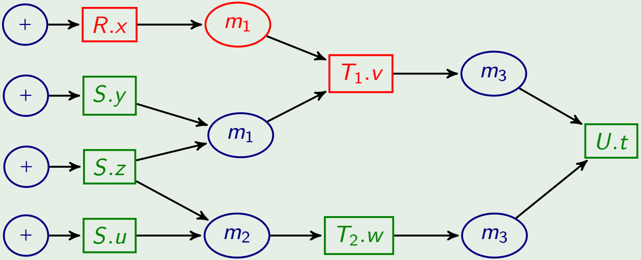


ILLINOIS INSTITUTE OF TECHNOLOGY

Example View Expressions

Example

`[] <- [R]`

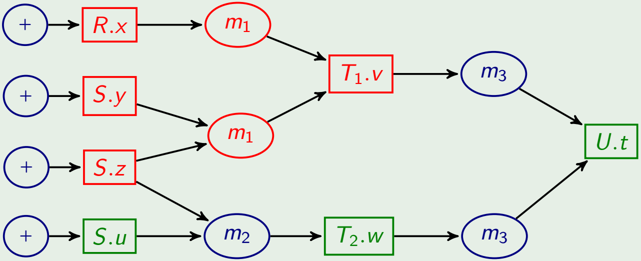


ILLINOIS INSTITUTE OF TECHNOLOGY

Example View Expressions

Example

[] <m_1 []

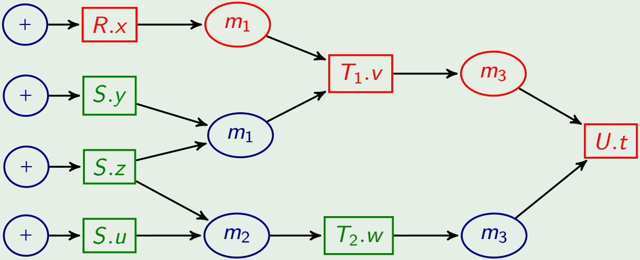


ILLINOIS INSTITUTE OF TECHNOLOGY

Example View Expressions

Example

```
[U] <-+ [R]
```



ProQL Graph Projection Syntax

```
FOR var_binding
WHERE where_expr
INCLUDE PATH incl_expr
RETURN return_expr
```

- `FOR`: bind variables using path expressions
- `WHERE`: logical expression over conditions:
  - Tuple nodes: Conditions on attribute values and relation name
  - View nodes: Test view name
- `INCLUDE PATH`: Paths to copy to output graph
  - Evaluated for each binding of variables that fulfill `WHERE`
- `RETURN`: Variables bindings to return
  - Each combination as tuple

Examples for Graph Projection Syntax

```
FOR [R $x]
INCLUDE PATH [$x] <-+ []
RETURN $x
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Examples for Graph Projection Syntax

```
FOR [R $x]
INCLUDE PATH [$x] <-+ []
RETURN $x
```

Return the subgraph containing all derivations of tuples in R from base tuples.

Examples for Graph Projection Syntax

```
FOR [O $x] <-+ [A $y]
INCLUDE PATH [$x] <-+ [$y]
RETURN $x
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Examples for Graph Projection Syntax

```
FOR [O $x] <-+ [A $y]
INCLUDE PATH [$x] <-+ [$y]
RETURN $x
```

Return the part of derivations of tuples in O that involve tuples in relation A.

Examples for Graph Projection Syntax

```
FOR [$x] <$p [], [$y] <- [$x]
WHERE $p=m1 OR $p=m2
INCLUDE PATH [$y] <- [$x]
RETURN $y
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Examples for Graph Projection Syntax

```
FOR [$x] <$p [], [$y] <- [$x]
WHERE $p=m1 OR $p=m2
INCLUDE PATH [$y] <- [$x]
RETURN $y
```

Find tuples that can be derived through mappings m_1 or m_2 and return all one-step derivations from those tuples.

ILLINOIS INSTITUTE OF TECHNOLOGY

Examples for Graph Projection Syntax

```
FOR [O $x] <-+ [$z], [C $y] <-+ [$z]
INCLUDE PATH [$x] <-+ [], [$y] <-+ []
RETURN $x, $y
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Examples for Graph Projection Syntax

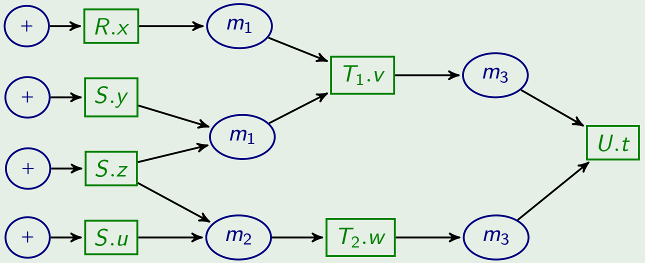
```
FOR [O $x] <-+ [$z], [C $y] <-+ [$z]
INCLUDE PATH [$x] <-+ [], [$y] <-+ []
RETURN $x, $y
```

Select tuples from O and C that have common provenance and return their derivations.

Example Evaluation

Example

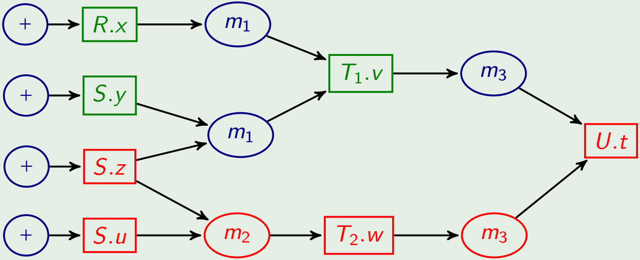
```
FOR [U $x] <-+ [$y], [$z] <m2 [$u]
WHERE [$y] <-+ [$z]
INCLUDE PATH [$x] <-+ [$u]
RETURN $x, $u
```



Example Evaluation

Example

```
FOR [U $x] <-+ [$y], [$z] <m2 [$u]
WHERE [$y] <-+ [$z]
INCLUDE PATH [$x] <-+ [$u]
RETURN $x, $u
```



$x	$u
U.t	S.z
U.t	S.u

Restrictions of Graph Projection

- No creation of new nodes or edges
- No negation
    - This is why we need the + nodes

Restrictions of Graph Projection

- No creation of new nodes or edges
- No negation
  - This is why we need the + nodes
  - Otherwise, how to return base tuples in provenance?
  - Because no notion of tuples **without** incoming edge

ILLINOIS INSTITUTE OF TECHNOLOGY

Annotation Computation

EVALUATE semiring OF

- Choose a semiring to evaluate the graph (provenance polynomial over)
  - **Trust**
  - **Bag multiplicity**
  - **Probability**
  - **Other Provenance Models**

ASSIGN EACH

- Assign semiring annotations to base tuple nodes
  - The ones connected to + view nodes
- Assign functions to views
  - A view function maps lists of annotations to annotations

Annotation Computation Syntax

```
EVALUATE semiring  OF {
   graph_projection
}
ASSIGNING EACH [leaf_node | view_node] variable {
  CASE case_condition ]: SET annotation
  [DEFAULT]: SET annotation
}
...
```

ILLINOIS INSTITUTE OF TECHNOLOGY

EVALUATE Semirings

Some semirings

Semiring	Values	$t + t'$	$t \times t'$
Trust	True/False	$t \wedge t'$	$t \vee t'$
Derivability	True/False	$t \wedge t'$	$t \vee t'$
Confidentiality level	Confidentiality	$max(t, t')$	$min(t, t')$
Weight/cost	base tuple weight	$t + t'$	$min(t, t')$

- And semirings representing other provenance models

ILLINOIS INSTITUTE OF TECHNOLOGY

ASSIGN EACH

```
ASSIGNING EACH [leaf_node | view_node(variables)] variable {
  CASE : SET annotation
  [DEFAULT]: SET annotation
}
case_condition: variable in relation [attr_expr ]
                       | variable = view [check_input]
```

- Choose a variable $\$x$ or view variable $\$p(\$x_1, \ldots, \$x_n)$ where $\$x_i$ are the arguments of view
- Assign annotation based on matching `CASE` expression
- ...if none matches $\Rightarrow$annotation from `DEFAULT`
- Each `CASE` check for membership in relation
- Optionally

Annotation Computation Example

```
ASSIGNING EACH $y {
  CASE $y in S AND S.a < 3: SET true
  DEFAULT: SET false
}
```

Annotation Computation Example

```
ASSIGNING EACH $p($x) {
  CASE $p = m1: SET false
  DEFAULT $x
}
```

ILLINOIS INSTITUTE OF TECHNOLOGY

Annotation Computation Example

```
EVALUATE derivability OF {
  FOR [O $x]
  INCLUDE PATH [$x] <-+ []
  RETURN $x
}
```

Annotation Computation Example

```
EVALUATE trust OF {
  FOR [O $x]
  INCLUDE PATH [$x] <-+ [] RETURN $x
}
ASSIGNING EACH leaf_node $y {
  CASE $y in C : SET true
  CASE $y in A AND $y.height >= 6 : SET false
  DEFAULT : SET true
}
ASSIGNING EACH mapping $p($z) {
  CASE $p = m4 : SET false
  DEFAULT : SET $z
}
```

Implementation

- Use relational representation of provenance graph
  - Reuse existing relations for tuple nodes
  - Add new relations for edges and view nodes
- Translate ProQL into SQL (datalog) queries

Relational Storage Scheme

- Tuple nodes are tuples in regular database tables
- Store view nodes and edges in addition relations
- Each relation stores one type of derivation for one view
  - Attributes for the input tuple keys
  - Attributes for output tuple key
- Similar to edge relations discussed for provenance storage

Example

```
SELECT b AS e FROM R JOIN S ON (R.a = S.c)
```

R

	a	b
r_1	1	2
r_2	3	4

S

	c	d
r_1	1	5
r_2	2	6

M_1

	e
t_1	2

$\mathbf{D}(m_1)$

a	c	e
1	1	2

Translation of ProQL into SQL

1. Build a **provenance schema graph** based on views
   - Independent of query
2. Match query to schema graph
   - Which nodes in schema graph may match path expressions
3. Create datalog program based on matching
4. Execute program using DBMS

ILLINOIS INSTITUTE OF TECHNOLOGY

1) Provenance Schema Graph

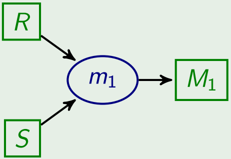
- Models possible provenance relationships among tuples
- One node for each view
- One node for each relation
- Directed edge from a relation to view
  - View accesses this relation
- Directed edge from view to relation
  - View generates this relation
- Property: Each path in the a provenance graph corresponds to path in schema graph
  - by mapping each tuple node to its relation
  - by mapping a view application node to the view node

GY

1) Provenance Schema Graph

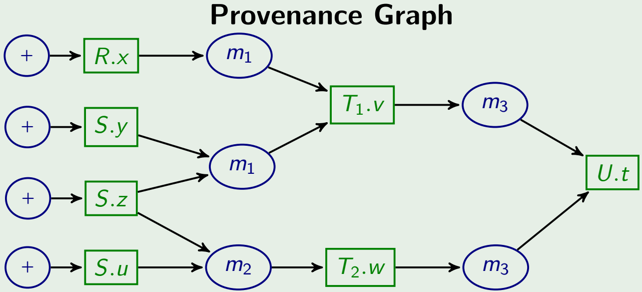
Example

```
CREATE VIEW M1
SELECT b AS e FROM R JOIN S ON (R.a = S.c)
```



Example Schema Graph

Example



Example Schema Graph

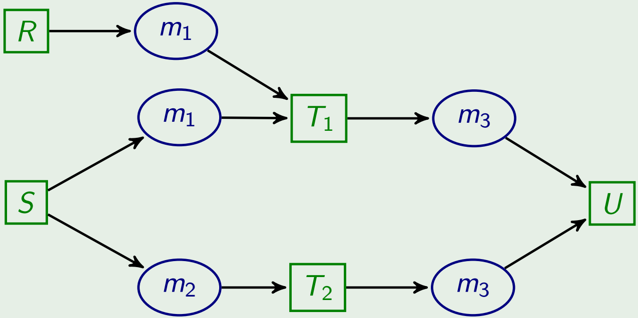
Example

```
CREATE VIEW T1 AS (
SELECT * FROM R
UNION
SELECT a.* FROM S a, S b WHERE a.x = b.y)

CREATE VIEW T2 AS (
SELECT a.* FROM S a, S b WHERE a.y = b.y)

CREATE VIEW U AS (
SELECT * FROM T1
UNION
SELECT * FROM T2
```

Example Schema Graph

Example

Provenance Schema Graph



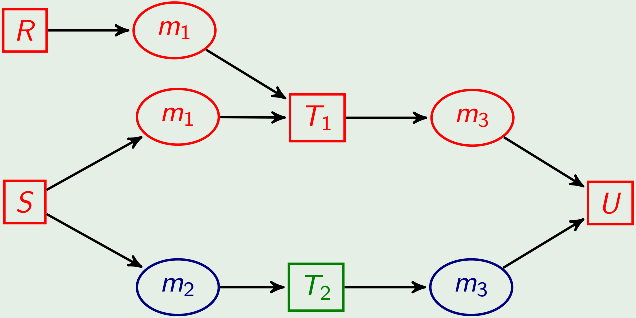
2) Matching Path Expressions

- For each path expression in the query
- Start with all relation nodes that match the start point
- Compute all potential paths in the schema graph that match the path expression
  - Restriction on views
  - Restriction on end points

Example Matching Path Expressions

Example

```
FOR [$z] <- [$y], [$y] <m1 [$x]
INCLUDE PATH [$z] <-+ [$x]
RETURN $x, $z
```



3) Generate Datalog Program

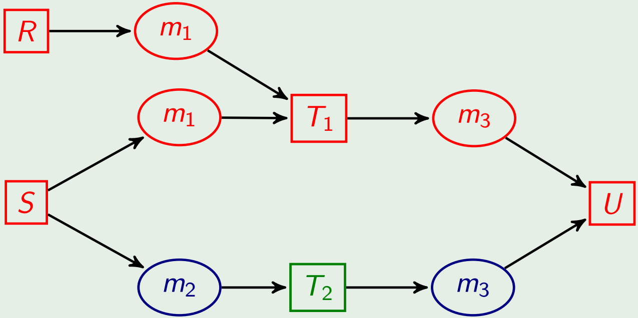
- Build rules for each path in schema graph for each variable
  - Edges are tuples from provenance tables
  - Joins between provenance tables are paths
- Integrate `WHERE` conditions
- Result tables
  - **Provenance tables**: store the edges in the result graph
  - **RETURN bindings**: store variable bindings from `RETURN` clause

ILLINOIS INSTITUTE OF TECHNOLOGY

Example Generate Datalog Program

Example

```
FOR [$z] <- [$y], [$y] <m1 [$x]
INCLUDE PATH [$z] <-+ [$x]
RETURN $x, $z
```



Example Generate Datalog Program

Example

```
FOR [$z] <- [$y], [$y] <m1 [$x]
INCLUDE PATH [$z] <-+ [$x]
RETURN $x, $z
```

Provenance Tables

Two tables for m_1: $m_1(r, t_1)$ and $m_1'(s_1, s_2, t_1)$
One table for m_2: $m_2(s_1, s_2, t_2)$
Two tables for m_3: $m_3(t_1, u)$ and $m_3'(t_2, u)$

Datalog Rules for RESULT

$$return(x, z) :- m_1(x, y) \wedge m_3(y, z)$$

$$return(x, z) :- m_1'(x, u, y) \wedge m_3(y, z)$$

$$return(x, z) :- m_1'(u, x, y) \wedge m_3(y, z)$$

Example Generate Datalog Program

Example

```
FOR [$z] <- [$y], [$y] <m1 [$x]
INCLUDE PATH [$z] <-+ [$x]
RETURN $x, $z
```

Provenance Tables

Two tables for m_1: $m_1(r, t_1)$ and $m_1'(s_1, s_2, t_1)$
One table for m_2: $m_2(s_1, s_2, t_2)$
Two tables for m_3: $m_3(t_1, u)$ and $m_3'(t_2, u)$

Datalog Rules for Result Graph Edges

$$m_1(x, y) :- m_1(x, y) \wedge m_3(y, z)$$

$$m_1'(x, u, y) :- m_1'(x, u, y) \wedge m_3(y, z)$$

$$m_3(x, z) :- m_1(x, y) \wedge m_3(y, z)$$

$$m_3(x, z) :- m_1'(x, u, y) \wedge m_3(y, z)$$

4) Execute over DBMS

- Add `CASE` constructs for `ASSINGING EACH` clause
- Determine how provenance polynomial structure using paths
- `UNION ALL` instances of paths
- Use aggregation functions to combine annotations
- If provenance graphs do not have cycles
  - Unfold the recursive datalog rules
  - ⇒Eliminates recursion

Example

Example

Datalog Rules for RESULT

$$return(x,z) :- m_1(x,y) \wedge m_3(y,z)$$
$$return(x,z) :- m'_1(x,u,y) \wedge m_3(y,z)$$
$$return(x,z) :- m'_1(u,x,y) \wedge m_3(y,z)$$

SELECT r AS x, u AS z FROM m_1, m_3 WHERE $m_1.t_1$ = $m_3.t_1$
UNION ALL
SELECT s_1 AS x, u AS z FROM m'_1, m_3 WHERE $m'_1.t_1$ = $m_3.t_1$
UNION ALL
SELECT s_2 AS x, u AS z FROM m'_1, m_3 WHERE $m'_1.t_1$ = $m_3.t_1$

GY

Discussion ProQL

- **New graph query language**
- Only for provenance **Retrieval**
- Input and Output are provenance graphs
- Selection of subgraphs and correlation of nodes (variable bindings in `RETURN`)
- Compute semiring annotations depending on use case
- Provenance Graph stored as "combined" edge-relation (keys of input tuples and output tuple key)
- Translation to datalog - matching path expressions to schema graph
  - Can result in large number of rules (recall Interval encoding)

Outline

1. Querying Languages for Provenance
   - Querying Provenance Overview
   - DBNotes' pSQL
   - Perm's SQL-PLE
   - ProQL
   - Color Algebra
   - Recap

Color Algebra

Overview

- Query language for annotation propagation (colors)
- Data model: Relational with annotations
  - Annotations on **blocks** of values spanning multiple attributes
- Language is an extension of relational algebra
  - Standard operators that propagate annotations
  - Operators that access annotations
- Annotation propagation always active

Data Model

Annotated Relations

- Annotations attached to sets of attributes (blocks) of a tuple
- Annotations are called colors
- Blocks can overlap
- Formally relations + $\mathcal{C}$ (colors) and function $\chi(t, Y) \rightarrow C$:
  - Maps a tuple t and a set of attributes Y from t
  - to a set C of colors (annotations) from $\mathcal{C}$

Example

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

(pink): {Manager}
(green): {CAD}
(blue): {Since 2000}

$$\chi(e_1, \{Name\}) = \{Manager\}$$
$$\chi(e_1, \{Salary\}) = \{CAD\}$$
$$\chi(e_3, \{Name, Salary\}) = \{Since\ 2000\}$$

Color Algebra Operators

- Operators map colored relations to colored relation
  - $R \rightarrow R'$
  - $\chi \rightarrow \chi'$

Operators with Annotation Propagation

- Projection
- Selection
- Cross Product (Join)

Operators Accessing Annotations

- Color Projection
- Color Selection
- Merge

Projection

- Projection $\pi_A(R)$
  - A is a set of attributes
- Standart relational projection for data
- Keep all annotations on blocks overlapping with A

Definition (Projection)

$\pi_{A_1,\ldots,A_k}(R)$

$$R' = \{t[A_1, \ldots, A_k] \mid t \in R\}$$

$$\chi'(t[A_1, \ldots, A_k], Y) = \bigcup_{Z \subseteq \mathbf{R}} \chi(t, Y \cup Z)$$

Projection

Example

$\pi_{Name}(Employee)$

Employee

	Name	Salary
e_1	Peter	$30
e_2	Helga	$50
e_3	Ann	$199

$$\chi(e_1, \{Name\}) = \{Manager\}$$
$$\chi(e_1, \{Salary\}) = \{CAD\}$$
$$\chi(e_3, \{Name, Salary\}) = \{Since\ 2000\}$$

Q

	Name
t_1	Peter
t_2	Helga
t_3	Ann

$$\chi(e_1, \{Name\}) = \{Manager\}$$
$$\chi(e_3, \{Name\}) = \{Since\ 2000\}$$

OF TECHNOLOGY

Selection

- Selection $\sigma_C(R)$
- C is an equality comparison like
  1. $A = a$ where A is attribute and a a constant
  2. $A = B$ where A and B are attributes
- Standart relational selection for data
- For (1) Keep all annotations on tuples $t \models C$
- For (2) Only keep colors c for exists blocks
  - Block containing A of color c
  - Block containing B of color c

ILLINOIS INSTITUTE OF TECHNOLOGY

Selection

Definition (Selection)

- $\sigma_{A=a}(R)$

$$R' = \{t \mid t \in R \land t.A = a\}$$
$$\chi'(t,Y) = \chi(t,Y)$$

- $\sigma_{A=B}(R)$

$$R' = \{t \mid t \in R \land t.A = t.B\}$$
$$\chi'(t,Y) = \begin{cases} \chi(t,Y) & \text{if } A, B \notin Y \\ \chi(t,Y) \cap \beta(t,A) \cap \beta(t,B) & \text{otherwise} \end{cases}$$
$$\beta(t,X) = \{c \mid \exists Y : X \in Y \land \chi(t,Y) = c\}$$

GY

Selection

Example

$\sigma_{Name=Nickname}(Names)$

Names

	Name	Nickname
n_1	Peter	Fatty
n_2	Helga	Rose
n_3	Ann	Ann

$\chi(n_1, \{Name\}) = \{Middle\ name\}$

$\chi(n_1, \{Nickname\}) = \{Why?\}$

$\chi(n_3, \{Name, Nickname\}) = \{Boring\}$

Q

	Name	Nickname
t_1	Ann	Ann

$\chi(t_1, \{Name\}) = \{Boring\}$

$\chi(t_1, \{Nickname\}) = \{Boring\}$

Cross Product

- Cross Product $R \times S$
- Standart relational cross product for data
- Keep all annotations

Definition (Cross Product)

$R \times S$ with color functions χ_R and χ_S

$$R' = \{t \times t' \mid t \in R \wedge t' \in S\}$$

$$\chi'(t, Y) = \begin{cases} \chi_R(\pi_{\mathbf{R}}(t), Y) & \text{if } Y \subseteq \mathbf{R} \\ \chi_S(\pi_{\mathbf{S}}(t), Y) & \text{if } Y \subseteq \mathbf{S} \\ \emptyset & \text{otherwise} \end{cases}$$

Cross Product

Example

Employee × *Names*

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

Names

	FirstName	Nickname
n_1	Peter	Fatty
n_2	Helga	Rose
n_3	Ann	Ann

$$\chi(e_1, \{Name\}) = \{Manager\}$$
$$\chi(n_1, \{NickName\}) = \{Mean\}$$
$$\chi(n_3, \{FirstName, Nickname\}) = \{Boring\}$$

Cross Product

Example

$Employee \times Names$

Q

	Name	Salary	FirstName	Nickname
t_1	Peter	\$30	Ann	Ann
t_2	Helga	\$50	Ann	Ann
t_3	Ann	\$199	Ann	Ann
	...	...	...	...

$$\chi(t_1, \{FirstName\}) = \{Manager\}$$
$$\chi(t_1, \{Name, Nickname\}) = \{Boring\}$$
$$\chi(t_2, \{Name, Nickname\}) = \{Boring\}$$
$$\chi(t_3, \{Name, Nickname\}) = \{Boring\}$$

ILLINOIS INSTITUTE OF TECHNOLOGY

Block Projection (Lower)

- Lower Block Projection $\pi^L_A(R)$
  - A is a set of attributes
- Keep only tuples that have at least one annotation on a superset of A
- Keep only annotations on blocks covering A

Definition (Block Projection (Lower))

$\pi^L_A(R)$

$$R' = \{t \mid t \in R \wedge \exists Y : A \subseteq Y \wedge \chi(t, Y) \neq \emptyset\}$$

$$\chi'(t, Y) = \begin{cases} \chi(t, Y) & \text{if } A \subseteq Y \\ \emptyset & \text{otherwise} \end{cases}$$

Block Projection (Lower)

Example

$\pi^{L}_{Name}(Employee)$

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

$\chi(e_1, \{Name\}) = \{Manager\}$
$\chi(e_1, \{Salary\}) = \{CAD\}$
$\chi(e_3, \{Name, Salary\}) = \{Since\ 2000\}$

Q

	Name	Salary
t_1	Peter	\$30
t_2	Ann	\$199

$\chi(t_1, \{Name\}) = \{Manager\}$
$\chi(t_2, \{Name, Salary\}) = \{Since\ 2000\}$

Block Projection (Upper)

- Upper Block Projection $\pi^U_A(R)$
  - A is a set of attributes
- Keep all tuples
- Keep only annotations on blocks contained in A

Definition (Block Projection (Upper))

$\pi^L_A(R)$

$$R' = R$$

$$\chi'(t, Y) = \begin{cases} \chi(t, Y) & \text{if } Y \subseteq A \\ \emptyset & \text{otherwise} \end{cases}$$

Block Projection (Upper)

Example

$\pi^{U}_{Name}(Employee)$

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

$\chi(e_1, \{Name\}) = \{Manager\}$

$\chi(e_1, \{Salary\}) = \{CAD\}$

$\chi(e_3, \{Name, Salary\}) = \{Since\ 2000\}$

Q

	Name	Salary
t_1	Peter	\$30
t_2	Helga	\$50
t_3	Ann	\$199

$\chi(t_1, \{Name\}) = \{Manager\}$

Block Selection

- Block Selection $\Sigma_c(R)$
- c is a color from $\mathcal{C}$
- Keep only tuples with at least one block of color c
- Keep only annotations of color c

Definition (Block Selection)

$\Sigma_c(R)$

$$R' = \{t \mid t \in R \land \exists Y : c \in \chi(t, Y)\}$$
$$\chi'(t, Y) = \chi(t, Y) \cap \{c\}$$

GY

Block Selection

Example

$\sigma_{Manager}(Names)$

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

$\chi(e_1, \{Name\}) = \{Manager\}$

$\chi(e_1, \{Salary\}) = \{CAD\}$

$\chi(e_3, \{Name, Salary\}) = \{Since\ 2000\}$

Q

	Name	Salary
t_1	Peter	\$30

$\chi(t_1, \{Name\}) = \{Manager\}$

OF TECHNOLOGY

Merge

- Merge $\mu_{X,Y}(R)$
- X and Y are sets of attributes from R
- Keep all tuples
- Merge annotations on blocks contained in X and Y

Definition (Merge)

$\mu_{X,Y}(R)$

$$
\begin{aligned}
R' =& R \\
\chi'(t, Z) =& \{c \mid \exists X_1, X_2 : Z = X_1 \cup X_2 \land X_1 \subseteq X \land X_2 \subseteq Y \\
& \land c \in \chi(t, X_1) \land c \in \chi(t, X_2)\}
\end{aligned}
$$

Merge

Example

$\mu_{\{Name\},\{Salary\}}(Employee)$

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

$\chi(e_1, \{Name\}) = \{Manager\}$

$\chi(e_1, \{Salary\}) = \{CAD\}$

$\chi(e_3, \{Name\}) = \{Since\ 2000\}$

$\chi(e_3, \{Salary\} = \{Since\ 2000\}$

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

$\chi(t_3, \{Name, Salary\}) = \{Since\ 2000\}$

GY

Union

- Union $R \cup S$ with color functions χ_R and χ_S
- Apply standard relational union to data
- Keep all annotations from both inputs

Definition (Union)

$R \cup S$

$$R' = R \cup S$$
$$\chi'(t, Y) = \chi_R(t, Y) \cup \chi_S(t, Y)$$

ILLINOIS INSTITUTE OF TECHNOLOGY

Implementation in **Mondrian**

Relational Representation

- Extend each relation R with additional attributes to store annotations
  - A boolean attribute bA for each attribute A of R to store block memberships
  - One attribute λ to store the actual annotation
- Duplicate tuples if necessary (Set semantics)

Query rewrite

- Rewrite color algebra expressions into SQL

OF TECHNOLOGY

Example Relational Representation

Example

Employee

	Name	Salary
e_1	Peter	\$30
e_2	Helga	\$50
e_3	Ann	\$199

$\chi(e_1, \{Name\}) = \{Manager\}$

$\chi(e_1, \{Salary\}) = \{CAD\}$

$\chi(e_3, \{Name, Salary\}) = \{Since\ 2000\}$

rep(Employee)

	Name	Salary	bName	bSalary	λ
e_1	Peter	\$30	1	0	Manager
e_1	Peter	\$30	0	1	CAD
e_2	Helga	\$50	0	0	-
e_3	Ann	\$199	1	1	Since 2000

GY

Rewrite Into SQL

- SQL translation for each operator
- Recursively translate algebra expression
- Execute resulting (single) SQL query over relational representation

Color Algebra Recap

- Query language for annotions relations
- Extension of relational algebra
- Annotation propagation always active
- Implementation
  - Relational representation
  - Rewrite into SQL
- No creation of new annotations
- Fixed options for propagating annotations

ILLINOIS INSTITUTE OF TECHNOLOGY

Outline

Recap

Motivation for Querying Provenance

- Size of Provenance Information
- Extract parts of interest from provenance

Challenges of Querying Provenance

- Different model for data and provenance
- Queries that span regular data and provenance
- Different access patterns

Recap

Language Design

- Support Retrieval and/or Generation features
- Extending the data query language
- . . . or Develop new language

Language Implementation

- Compile into data query language
- Create new or modify execution engine

Recap

Covered Languages

- **pSQL**
  - Generation and retrieval language for annotations
  - Extension of subset of SQL compiled into SQL
  - Annotation propagation always active
  - **DBNotes**
- **SQL-PLE**
  - Generation language for provenance
  - Retrieval using SQL
  - Extension of SQL
  - **Perm**
- **ProQL**
  - Retrieval language for provenance graphs ($\mathbb{N}[I]$)
  - New query language complied into SQL
  - **Orchestra**

Recap

Covered Languages

- **Color Algebra**
  - Generation and retrieval language for annotations
  - Extension of relational algebra compiled into SQL
  - Annotation propagation always active
  - **Mondrian**

ILLINOIS INSTITUTE OF TECHNOLOGY

Literature I

Zachary G. Ives, Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Val Tannen, Partha Pratim Talukdar, Marie Jacob, and Fernando Pereira.
The ORCHESTRA Collaborative Data Sharing System.
SIGMOD Record, 37(2):26–32, 2008.

G. Karvounarakis, Z.G. Ives, and V. Tannen.
Querying data provenance.
In Proceedings of the 2010 international conference on Management of data, 951–962, ACM, 2010.

Todd J. Green, Grigoris Karvounarakis, Nicholas E. Taylor, Olivier Biton, Zachary G. Ives, and Val Tannen.
ORCHESTRA: Facilitating Collaborative Data Sharing.
In SIGMOD '07: Proceedings of the 33th SIGMOD International Conference on Management of Data, 2007.

Boris Glavic and Gustavo Alonso.
The Perm Provenance Management System in Action.
In Proceedings of the 35th ACM SIGMOD International Conference on Management of Data (SIGMOD) (Demonstration Track), 1055-1058, 2009.

Boris Glavic and Gustavo Alonso.
Provenance for Nested Subqueries.
In Proceedings of the 12th International Conference on Extending Database Technology (EDBT), 982-993, 2009.

Boris Glavic and Gustavo Alonso.
Perm: Processing Provenance and Data on the same Data Model through Query Rewriting.
In Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE), 174-185, 2009.

ILLINOIS INSTITUTE OF TECHNOLOGY

Literature II

Boris Glavic.
Perm: Efficient Provenance Support for Relational Databases.
PhD thesis, University of Zurich, 2010.

Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya.
DBNotes: a Post-it System for Relational Databases based on Provenance.
In SIGMOD '05: Proceedings of the 31th SIGMOD International Conference on Management of Data, 942–944, 2005.

Deepavali Bhagwat, Laura Chiticariu, Wang-Chiew Tan, and Gaurav Vijayvargiya.
An Annotation Management System for Relational Databases.
The VLDB Journal, 14(4):373–396, 2005.

Floris Geerts, Anastasios Kementsietsidis, and Diego Milano.
MONDRIAN: Annotating and Querying Databases through Colors and Blocks.
Technical report, University of Edinburgh, 2005.

Floris Geerts, Anastasios Kementsietsidis, and Diego Milano.
iMONDRIAN: A Visual Tool To Annotate and Query Scientific Databases.
In EDBT '06: Proceedings of the 9th International Conference on Extending Database Technology (demonstration), 1168–1171, 2006.

ILLINOIS INSTITUTE OF TECHNOLOGY